

## SOFTWARE-TO-HARDWARE COMPILER

5 This application claims the benefit of United States provisional application No. 60/223,471, filed August 7, 2000, which is hereby incorporated by reference herein in its entirety.

Background of the Invention

10 This invention relates to a high-level programming language compiler, and more particularly, this invention relates to a high-level programming language compiler that converts a high-level programming language into hardware constructs.

15 Programmable logic provides many benefits in the design and manufacture of hardware and software systems. For example, efficiencies with respect to cost and time are known benefits. Programmable logic allows designers to avoid having to hard-wire circuits. Designers typically use specialized hardware design software to create logic circuits by interconnecting various types of logic gates, memory, etc on a computer screen.

20 Some known software packages allow the designer to write code in which various hardware constructs are specified. These software packages are typically

specially designed for use with particular programmable logic hardware.

These tools do not, however, reach the ease of use and range of compatibility as do high-level language software compilers. For example, JAVA and C++ are widely used high-level languages that can be used to create various types of software constructs, such as loops, conditionals, functions, etc. with ease. These software constructs are compiled into a software program that can be executed on a computer's microprocessor (i.e., using the built-in instruction set). There does not exist any such high-level language compiler that can compile pure software constructs (i.e., that are transparent with regard to hardware) into programmable logic configuration data.

Moreover, the tools that are available for generating programmable logic in hardware typically lack the ability to produce efficient circuitry. For example, most of these tools implement software variables as registers in hardware. This leads to the necessity of having to multiplex every write to a register. Because multiplexers are large and slow in programmable logic, this implementation inherently leads to inefficiencies and makes retiming the circuit difficult.

One of the advantages of using hardware over software implementations is that hardware offers a greater degree of parallelism than does software. Software constructs typically execute in a sequential order. For example, loop iterations execute one after the other and conditional statements are evaluated before deciding whether to execute an ELSE construct or a THEN construct. The parallel nature of hardware has the potential to provide for more efficient execution and decision-making than that offered by software. However, present-day programmable logic compilers and tools do not

generate programmable logic that takes full advantage of its parallel nature.

It would therefore be desirable to have a software-to-hardware compiler that maps pure software  
5 constructs into hardware constructs implemented in programmable logic.

It would further be desirable to be able to generate hardware in programmable logic that is optimized for parallelism.

## 10 Summary of the Invention

It is therefore an object of the present invention to provide a software-to-hardware compiler that maps pure software constructs into hardware constructs implemented in programmable logic.

15 It is a further object of the present invention to generate hardware in programmable logic that is optimized for parallelism.

These and other objects are accomplished in accordance with the principles of the present invention  
20 by providing a compiler that converts a high-level programming language into hardware constructs.

A software-to-hardware compiler parses through code written in a high-level programming language that defines pure software constructs. That is, hardware  
25 configurations and constructs are not included anywhere in the code. The software constructs are mapped directly into hardware constructs such that the program defined by the software constructs executes substantially the same in the hardware implementation with respect to doing what  
30 the program was intended to do.

The hardware constructs are implemented in programmable logic using programmable logic resources such as programmable logic devices, field programmable gate arrays, or any other suitable programmable logic

resources. In order to avoid the use of too many large multiplexors, variables may be implemented in hardware as a set of wires. One wire of a variable is used to indicate whether that value has been computed (i.e., is  
5 valid for use) while the remainder of the wires indicate what the value is.

Based on this mapping, expressions, arrays, pointers, and any other suitable data structures may likewise be implemented using wires (i.e., as opposed to  
10 registers). In one suitable approach, wires and registers may both be used concurrently to represent variables (e.g., for retiming optimizations, off-chip communications, etc.).

The software-to-hardware compiler of the  
15 present invention may perform optimization analysis at a later stage of a software-to-hardware compilation. Based on the optimization analysis, suitable modifications to the resultant circuit may be made to maximize data flow, increase the clock's frequency, or optimize the hardware  
20 in any other suitable way. For example, the software-to-hardware compiler may analyze the critical path of the hardware and place registers at points in the circuit to maintain a balance in the circuit.

Programmable logic is divided into blocks of  
25 operations. The blocks are coupled to one another using a programmable interconnect. A control flow is passed into every block that specifies whether and how the block is to execute. The control flow may be used to make run-time decisions by the hardware in terms of parallel  
30 execution of blocks.

Parallel execution may include speculation, pipelining (e.g., loop-pipelining), shared blocks, or any other suitable form of parallel execution.

Speculation may take place in the situation  
35 where there is an uncertainty with regard to whether a

block will be executed. If execution of the block will not result in relevant changes in the system, then the block may be executed in parallel with other blocks. For example, the THEN part and ELSE part of a conditional  
5 block may be executed speculatively (i.e., in parallel) assuming there are no consequences in doing so.

The software-to-hardware compiler of the present invention may generate hardware that is capable of sharing blocks (i.e., to implement different parts of  
10 the software program). The shared blocks may be explicitly defined by a user using functions in the high-level language code. In another suitable approach, the software-to-hardware compiler may be given the ability to recognize common expressions and implement them as shared  
15 blocks in the hardware.

Loop-pipelining allows more than one iteration of a loop to be executed simultaneously. A pertinent state register may be used to store and update intermediate values that are used in the loop iterations.  
20 The intermediate values are eventually merged into a final output of the loop.

#### Brief Description of the Drawings

The above and other objects of the present  
25 invention will be apparent upon consideration of the following detailed description, taken in conjunction with the accompanying drawings, in which like reference characters refer to like parts throughout, and in which:

FIG. 1 is a block diagram showing how the  
30 software-to-hardware compiler generates hardware in accordance with the present invention;

FIG. 2 is a block diagram of a program having software constructs and hardware constructs;

FIG. 3 is a block diagram of a program having only software constructs in accordance with the present invention;

FIG. 4 shows how a variable and its value is mapped into hardware in accordance with the present invention;

FIG. 5 shows how a constant and its value is mapped into hardware in accordance with the present invention;

FIG. 6 is a flow chart of illustrative steps involved in computing a value of a variable in accordance with the present invention;

FIG. 7 is a schematic of an illustrative assignment operator in accordance with the present invention;

FIG. 8 shows an illustrative unary operator in accordance with the present invention;

FIG. 9 shows an illustrative binary operator in accordance with the present invention;

FIG. 10 is a schematic of an illustrative addition operator in accordance with the present invention;

FIG. 11 is a flow chart of illustrative steps involved in optimizing a circuit by adding registers in accordance with the present invention;

FIG. 12 shows a simplified block in accordance with the present invention;

FIG. 13 shows the amalgamation of two blocks into a single block in accordance with the present invention;

FIG. 14 shows how speculation may be carried out on an IF-THEN-ELSE construct in accordance with the present invention;

FIG. 15 is a flow chart of illustrative steps involved in executing an operation speculatively in accordance with the present invention;

FIG. 16 is an illustrative environment in  
5 accordance with the present invention;

FIG. 17 shows the relationship between control flow and an operator in accordance with the present invention;

FIG. 18 is a flow chart of illustrative steps  
10 involved in coupling the input to a block or blocks to the output of that block or blocks in accordance with the present invention;

FIG. 19 is a diagram illustrating how shared blocks may be defined in accordance with the present  
15 invention;

FIG. 20 is a flow chart of illustrative steps involved in sharing a resource in the same context in accordance with the present invention;

FIG. 21 is a schematic diagram illustrating how  
20 sharing may be implemented in accordance with the present invention;

FIG. 22 is a schematic diagram of an illustrative sync block in accordance with the present invention;

FIG. 23 is a schematic diagram of an illustrative arbitration scheme in accordance with the present invention;

FIG. 24 is a schematic diagram illustrating how an array may be read in accordance with the present  
30 invention;

FIG. 25 is a schematic diagram illustrating how an array may be written to in accordance with the present invention;

FIG. 26 is a schematic diagram illustrating how an indirect pointer read may be implemented in accordance with the present invention;

FIG. 27 is a schematic diagram illustrating how an indirect pointer write may be implemented in accordance with the present invention;

FIG. 28 is a schematic diagram of an illustrative IF-THEN-ELSE hardware construct in accordance with the present invention;

FIG. 29 is a schematic diagram of an illustrative decide block in accordance with the present invention;

FIG. 30 are schematic diagrams of illustrative join, flowjoin, and joinctl blocks in accordance with the present invention;

FIG. 31 is a schematic diagram of an illustrative loop mapped into hardware in accordance with the present invention;

FIG. 32 are schematic diagrams of illustrative commitctl and commit blocks in accordance with the present invention;

FIG. 33 are schematic diagrams of illustrative merge and override blocks in accordance with the present invention;

FIG. 34 is a diagram of a simplified version of the circuit of FIG. 25 in accordance with the present invention;

FIG. 35 is a decision diagram showing how to determine if pipelining is possible in a loop in accordance with the present invention;

FIG. 36 is a block diagram illustrating loop-pipelining in accordance with the present invention;

FIG. 37 is a schematic diagram of an illustrative FIFO that may be read to provide an input in accordance with the present invention;



FIG. 38 is a schematic diagram of an illustrative FIFO to which data may be written in accordance with the present invention;

FIG. 39 is a block in accordance with the present invention;

FIG. 40 is an illustrative block that may be used to implement a greatest common divisor program in accordance with the present invention;

FIG. 41 is a schematic diagram of an illustrative circuit that may be used to implement the block of FIG. 34 in accordance with the present invention;

FIG. 42 is a schematic diagram the circuit if FIG. 35 shown with synthesized sub-blocks in accordance with the present invention;

FIG. 43 is a block diagram illustrating how communication may be accomplished between platforms across a channel in accordance with the present invention;

FIG. 44 shows an illustrative packet specification in accordance with the present invention;

FIG. 45 is a block diagram of an illustrative shared memory-based channel implementation for software/hardware communication in accordance with the present invention;

FIG. 46 is a block diagram of an illustrative bus-based channel implementation for software/hardware communication in accordance with the present invention;

FIG. 47 is a super-packet in accordance with the present invention;

FIG. 48 is a block diagram illustrating how shared memory may be used to provide communication between hardware and software in accordance with the present invention;

FIG. 49 is a stream in accordance with the present invention;

FIG. 48 is a block diagram illustrating how shared memory may be used to provide communication between hardware and software in accordance with the present invention;

FIG. 50 is a schematic diagram of an illustrative circuit for generating stream data for one or more FIFO's in accordance with the present invention;

FIG. 51 is a schematic diagram of an illustrative stream merge block in accordance with the present invention;

FIGS. 52 and 53 are schematic diagrams for two illustrative implementations for buffering a stream in accordance with the present invention;

FIG. 54 is a schematic diagram of an illustrative circuit for allowing a merged stream to pass through input/output pins in accordance with the present invention;

FIG. 55 is a schematic diagram of a circuit used on a remote platform to receive data from a data channel in accordance with the present invention; and

#### Detailed Description of the Preferred Embodiments

The present invention provides application programmers, engineers, and designers with the ability to use a high-level programming language (e.g., C, C++, JAVA, COBOL, Fortran, LISP, BASIC, Pascal, or any other suitable high-level programming language) to configure programmable logic resources (e.g., a programmable logic device (PLD), a field programmable gate array (FPGA), or any other suitable programmable logic resource).

For brevity and clarity, the functions of the present invention are described primarily in terms of a software-to-hardware compiler. This is merely an illustrative abstraction of the present invention. For

example, in another suitable embodiment, the present invention may be described in terms of a parser or any other suitable utility that may be used to convert high-level code into configuration data for one or more  
5 programmable logic resources.

The goals of the present invention are accomplished by creating mappings from high-level programming language constructs to hardware constructs. For example, an addition instruction in a programming  
10 language is mapped to a set of gates that implement addition in hardware. The mappings may be complete mappings. That is, every software construct may be mapped to a corresponding hardware construct. This avoids having to take into account hardware when writing  
15 software programs. Thus, any software program may be translated into hardware without the need to resort to writing some portions in a hardware language.

FIG. 1 is a block diagram illustrating how the software-to-hardware compiler of the present invention is  
20 used to generate hardware. Program 21 may be written in a high-level programming language. It is parsed and analyzed by software-to-hardware compiler 23, which generates configuration data. The configuration data may be any suitable configuration data that is used to  
25 program programmable logic resources 25. Thus, software-to-hardware compiler 23 generates hardware via programmable logic resources 25 using software-based program 21. It will be understood that other components may be introduced into the arrangement of FIG. 1. For  
30 example, debuggers, interfaces (i.e., either software or hardware), intervening hardware components, or any other suitable component may be introduced.

FIG. 2 shows one way in which hardware may be generated using a high-level programming language.

35 Program 11 is written using software constructs 13 and

hardware constructs, codes, tags, etc. 15. In this embodiment, the author of program 11 must acknowledge the hardware on which program 11 will execute. The software-to-hardware compiler of the present invention, may use  
5 the approach shown in FIG. 1. In another embodiment, the software-to-hardware compiler of the present invention may compile programs such as program 17 of FIG. 3 by allowing the program author to provide only software constructs 19 -- without the need to specify any  
10 hardware-related information.

Software constructs 19 may include any suitable software-based constructs, such as commands, expressions, data structures, modules, variables, or any other suitable high-level language software constructs.

15 Software variables, as used herein, refer to the commonly accepted definition in the art (i.e., value and expression placeholders). Software variables may hold any suitable types of data, such as integers, floating point numbers, characters, arrays, pointers, or  
20 any other suitable types of data.

Software may be regarded as a sequence of nested "blocks." The software-to-hardware compiler of the present invention translates each block into hardware, connecting blocks together by attaching the  
25 values of variables at the output of preceding blocks to the values of the corresponding variables at the input to subsequent blocks.

The software-to-hardware compiler of the present invention represents the value of a variable at a  
30 particular point in the software program as a set of wires in hardware. The value of the same variable at different points in the software program may be represented by different sets of wires.

During the execution of a software program, the  
35 value of a software variable may change. At any given

point in the program, the value of a software variable is either computed or not computed (i.e., the variable is either valid for use or not valid for use, respectively). Loops and sharing may break this rule in a limited way.

5           At any given point in the execution of a software program, there are a set of variables that are in scope. An "environment" is a mapping from those variable names to the wires representing their values at any given point in the execution. The environment need  
10 not change unless a new variable comes into scope.

FIG. 4 shows an illustrative representation of the value of a software variable in hardware 10. N data wires 14 are used to represent the value itself, and an additional wire 12 indicates whether the value is  
15 computed. For purposes of brevity and clarity, the values of variables are primarily illustrated herein as bold wires. For example, variable 10 is identical to variable 16. The bold wire of variable 16 may include the N+1 wires of variable 10.

20           A variable that only ever has one value is termed constant. In hardware, a constant is represented as a set of wires having a particular value and an additional wire indicating that the value is always computed. FIG. 5 illustrates this representation of a  
25 constant 18. Wires 20 are used to hold the constant value. Wire 22 indicates that the constant is computed.

Assuming no loops or sharing, the software-to-hardware compiler of the present invention may ensure that once the value of a variable at a particular point  
30 in the execution of the program is computed, its value will not change. That is, the compiler generates hardware that ensures that this rule is not broken. FIG. 6 is a flow chart of illustrative steps reiterating this process. At step 31, the hardware generated by the  
35 software-to-hardware compiler computes the value of a

variable at a particular point in the execution of a program. At step 33, hardware generated by the software-to-hardware compiler ensures that the value computed does not change for the duration of the execution of the  
5 program. One of the advantages of this feature of the software-to-hardware compiler is the ability to make hardware control circuitry very small. Applying retiming optimizations is made easier as well.

In a software program, execution proceeds by  
10 performing operations (e.g., arithmetic operations, movement of data, generation of data, etc.). Some operations may affect the value of variables. The software-to-hardware compiler of the present invention may be given the ability to implement software operations  
15 into hardware. An operation in hardware takes wires representing the value of variables in the current environment as input, and has additional wires representing the values of variables as output. The environment is updated so that the variable names to  
20 which the operation assigns its results map to the wires used to store its new value. FIG. 7 illustrates this technique. Operator 56 assigns the value of one variable (i.e., represented by wires 60) to another variable (i.e., represented by wires 58). This may be done by  
25 replacing the representation of the assignee variable from wires 58 to wires 62. Thus, subsequent reads to "x" use wires 62 rather than wires 58.

Operations may occur in large expressions in which variables are not directly updated. In these  
30 cases, new variable names may be used to represent intermediate results. By keeping track of how values are computed, it may be possible to identify when two variables compute the same value, and hence can be "merged," reducing hardware resources being used.

The value of the variable at the output of an operation is deemed computed when all the values at the input of the operation, upon which the output depends, are computed.

5           Operations may be represented in hardware using purely asynchronous logic. In another suitable approach, operations may be represented in hardware using pipelined circuitry. If using pipelined circuitry, output data may not become computed until after some number of clock  
10 cycles. Different outputs may even have different delays. The software-to-hardware compiler may ensure that the overall functionality of the hardware is the same irrespective of delays that the operations may have.

FIGS. 8 and 9 show illustrative unary and  
15 binary operators 26 and 34, respectively. In the case of a unary operation, input data 24 may be used by unary operator 26 to produce output data 28. In the case of a binary operation, input data 30 and 32 may be used by binary operator 34 to produce output data 36. The  
20 operators of FIGS. 8 and 9 are merely illustrative. Any other suitable number of inputs and outputs may be implemented for any suitable corresponding operator.

FIG. 10 shows an illustrative addition  
operator 38 that may be implemented in hardware using the  
25 software-to-hardware compiler. Value wires 48 and 50 correspond to the values of addends of an addition operation. Value wires 48 and 50 may be inputs to a sub-addition operator 40 (i.e., used to determine the value of the sum of the addends). Defined wires 44 and 48  
30 indicate whether the addends are valid. If at AND gate 42, addition operator 38 determines that the addends are valid, then defined wire 54 of the output of addition operator 38 is also valid. Wires 54 and 52 correspond to the expression of the sum of the inputs.

In one suitable approach, the software-to-hardware compiler need not add pipelining to operations initially. Rather, as shown in FIG. 11, at step 65 a late stage performance optimization phase may run an analysis on the software to be compiled. At step 67, based on this analysis, the software-to-hardware compiler may determine various optimizations to make to the hardware to make it more efficient (e.g., decrease the time-critical path of the whole design). For example, the compiler may determine that the addition of registers (i.e., at step 69) at particular points will allow the hardware to be clocked at a higher frequency. This may result in an overall increase in data throughput, despite the extra register stage. In programmable logic, registers are closely associated with asynchronous logic blocks, and may often be added without incurring a size penalty. Any other suitable optimizations may be made by the software-to-hardware compiler at this late stage.

Reading a variable may be accomplished by coupling a read port of an operator to the current set of wires representing the variable.

Blocks are another representation of software in the hardware context. Programmable logic is composed of a number of basic blocks of hardware coupled together with programmable interconnect. A block takes wires from a single input environment and generates wires for a single output environment. FIG. 12 illustrates this process. Environment 74 is taken as an input by block 78. Block 78 may perform any suitable operations using environment 74 to produce the output -- environment 76.

FIG. 12 is merely an illustrative and simplified representation of a block and its inputs and outputs. It will be apparent that additional control signals are needed.



A block is composed of operations that may be arranged such that there are no looping dependencies. Because a block can modify control flow, whereas an operation cannot, a block is not necessarily a composite  
5 operation.

Sequential blocks may be amalgamated into a single block (without introducing an extra hierarchy). This is illustrated in FIG. 13 in which sequential blocks 80 and 82 are amalgamated into block 84. For  
10 reasons to be introduced later (e.g., speculation), sequential blocks do not necessarily execute in sequential order. Control flow dictates the order in which operations should be executed.

In a sequential program (i.e., in software),  
15 control flow is generally sequential. Exceptions occur when jumps or conditional instructions are encountered. In these cases it is necessary to wait for the control flow to specify which operations will be executed and which operations will not be executed.

20 Hardware allows a large number of operations to be executed simultaneously. The software-to-hardware compiler of the present invention uses a run-time control flow system to maximize the number of operations that can be executed simultaneously (while ensuring that execution  
25 behaves as though the operations were executed sequentially). In one suitable approach, operations may be executed even though it is not guaranteed that their results will be needed. This form of execution is called speculative execution, and may only be performed on  
30 operations whose changes can be undone.

FIG. 14 shows how speculative execution can be used to reduce the latency of a conditional statement (e.g., IF-THEN-ELSE) where the condition takes several clock cycles to evaluate. In approach 86, after  
35 conditional 90 is evaluated, depending on the result of

conditional 90, either a THEN-part 92 may be executed or an ELSE-part 94 may be executed. In approach 88, using speculative execution, conditional part 96, THEN-part 98, and ELSE-part 100 may be executed simultaneously.

5 Depending on the result of conditional 96, the result of either THEN-part 98 or ELSE-part 100 is thrown away. As illustrated, the number of clock-cycles used for speculative execution is less than using sequential execution.

10 FIG. 15 shows the general process for speculation performed by the hardware generated by the software-to-hardware compiler. At step 93, hardware generated by the software-to-hardware compiler determines whether a particular operation may execute speculatively  
15 with one or more other operations. If the operation may be executed speculatively, then, at step 95, the hardware executes that operation speculatively with the one or more other operations.

Hardware control flow is synthesized using a  
20 special set of control flow wires that are used to enable or disable operations within blocks. For a given operation, control flow may specify one of four states, outlined in the table below:

25

| Control flow | Description   |
|--------------|---|
| 10           | Commit: Control flow is guaranteed to reach this operation  |
| 01           | Cancel: Control flow is guaranteed not to reach this operation  |
| 00           | Speculate: Control flow may nor may not reach this operation (allows speculation - i.e., execution of non-mutable operations) |
| 11           | Invalid state   |

It will be understood that even though control flow is guaranteed to reach an operation, that operation may still have to wait for valid input variables.

Environments include references to the current  
5 value of the control flow. FIG. 16 shows an illustrative environment 72. Environment 72 includes variable value wires 64, array values wires 66, expression value wires 68, and control flow wires 70.

The hardware ensures that control flow  
10 maintains the following invariants: control values may only change from (00) to (10), or from (00) to (01) (i.e., control flow cannot change its mind and revoke a previous guarantee); all speculation must eventually end with a guarantee; and when control flow is cancelled, it  
15 does not matter whether the data is valid or not. This latter rule will enable further reductions in the size of the control logic needed.

Some operations may irreversibly affect the state of the hardware (e.g., a RAM write or a FIFO  
20 "push"). With few exceptions, these operations cannot be executed speculatively. Operations that can affect the state of the hardware are termed mutable operations.

Under most circumstances, mutable operations may not be executed speculatively. Extra control logic  
25 may be added to ensure that mutable operations do not erroneously execute speculatively. The control flow signals specify when mutable operation is permitted, but consistent with the definition of an operation, no operation is able to change the control flow as shown in  
30 FIG. 17. Control flow 102 may be used by operation 104, but control flow 102 cannot be altered by operation 104.

Mutable operations should generally not execute more than once. In one suitable approach, extra control logic may be added to ensure this.

There may be times when a mutable operation may safely execute speculatively. For example, when it is known that the state being overwritten is not needed (e.g., when initializing an array), a mutable operation  
5 may execute speculatively. For purposes of brevity and clarity, the present invention is primarily described herein without taking these cases in account. It will be understood that these cases may be taken into account when implementing the present invention.

10           Loops require that the input and output of a block (or set of blocks) are coupled to one another. The software-to-hardware compiler may perform this coupling without generating an asynchronous loop. In one suitable approach, the outputs of a block may be registered, the  
15 block may be reset, and the registered values may then be supplied to the input of the block. This approach may be used in any circumstance where the input and output of a block (or set of blocks) are coupled to one another. FIG. 18 illustrates the process. At step 109, one or  
20 more outputs of one or more blocks are registered. At step 111, the block or blocks are reset. At step 113, the registered values are supplied to the input of the one or more blocks.

Because the inputs and outputs of a block have  
25 different environments, they, themselves, may be different. Thus, the different environments may be merged in accordance with the present invention, and this is done by augmenting the output environment by computing the additional sub-expressions needed by the input  
30 environment.

The software-to-hardware compiler generates every block with a reset input that is used to reset all the states within that block at the next clock edge. After a block is reset, all values of variables within  
35 that block return to their uncomputed state, and all

control flow signals are reset. In one suitable approach, reset should only be used when all the operations within the block have executed.

As described thus far, the software-to-hardware  
5 compiler of the present invention synthesizes a unique hardware equivalent for every operation in the software source language. In one suitable approach, common sub-expressions may be implemented using the same hardware resources in a way that is transparent to users.

10 In another suitable approach, the software-to-hardware compiler may provide users with the ability to explicitly specify that a block of hardware should be reused. One way in which this may be accomplished, while maintaining transparency, is through the use of  
15 functions.

These two approaches are illustrated in  
FIG. 19. Software functions 115 that are explicitly  
defined by the author of a software program may be  
regarded by the software-to-hardware compiler as a shared  
20 block 119. Shared blocks 119 may also be defined by the software-to-hardware compiler using automatic,  
transparent determinations 117 that may be based on, for  
example, an optimization analysis. Approaches 115  
and 117 may be used independently or in combination with  
25 one another.

Every location within a context that requires  
the use of a shared block will be termed a "usage  
instance." As used herein, a "context" is the present  
carpeting of computed data and control guarantees. Data  
30 that is between a context and not being computed (i.e.,  
in the process of being computed) is referred to herein  
as an "execution frontier." Every usage instance can  
only use a shared block once per context. However,  
different usage instances may require the same shared  
35 operation to be reused within the same context.

This form of sharing may be made possible by adding latched outputs and resetting the shared block (i.e., making the shared block ready for use by another instance).

5           FIG. 20 shows how a resource can be shared in the same context. At step 114, an exclusive lock on the operation is acquired. In the case of multiple simultaneous requests, an arbitrator may determine who acquires the lock. At step 116, the shared operation's context is then integrated into the current context at the appropriate point. Once execution of the operation has completed, then at step 118, the results are held (on a per usage instance basis), so that the shared operation, as well as the usage instance's lock, can be released. Once a usage instance's lock is released at 10 step 120, it cannot be used further within the same context (i.e., until a context reset).

FIG. 21 is a circuit that illustrates how sharing may be implemented in accordance with the present invention. Every calling instance 124 can only request 20 shared block 126 when all the variables needed are valid., and the control flow state is "commit".

To account for the simultaneous usage requests that is inherent with parallelism and speculation, 25 arbitrator 128 may be used. Arbitrator 128 decides which instance 124 should be granted exclusive use of shared block 126. Once shared block 126 has finished (i.e., when all the output variables are valid, and the control flow is in its commit state), the output environment is latched in the appropriate hold blocks (e.g., hold 30 blocks 130). Arbitrator 128 then resets shared block 126. Thus, shared block 126 is then ready for use by the next instance. Arbitrator 128 notes which instances 124 have already be granted access, and prevents them from 35 accessing shared block 126 again.

If there are no "jumps" or exceptions from within a block, shared block 126 is guaranteed to finish once started. Hence, control flow from each instance 124 may be passed directly through to holding block 130.

5       Some variables may remain invariant within a block (i.e., they do not change). These invariant variables may be passed directly to holding block 130, and arbitrator 128 does not need to wait for these variables to become valid before allocating shared  
10   block 126.

There are some circumstances when a shared block can be allocated to a speculative instance. In general, there is a danger of deadlock occurring. However, in the case of blocks that do not contain any  
15   mutable operations, deadlock may be guaranteed not to occur.

Sync block 132 waits for all variables in an instance's environment to become computed (i.e., valid). Sync block 132 may be implemented as illustrated in  
20   FIG. 22 with the accompanying specification as follows:

Sync

| Control flow | All variables | Output        |
|--------------|---------------|---------------|
| Commit       | Undefined     | NotReady      |
| Commit       | Defined(X)    | CommitDone    |
| Cancel       | Undefined     | NotReady      |
| Cancel       | Defined(X)    | NotReady      |
| Speculate    | Undefined     | NotReady      |
| Speculate    | Defined(X)    | SpeculateDone |

Hold blocks 130 register the shared resources output variables when hold blocks 130 receive a SpeculateDone or CommitDone signal.

5       Usage instances may request a shared operation

when one of two particular conditions are met. First, when the instance's environment's control flow is committed, and all the data needed is available. Second, if no instances satisfy the first condition, then  
5 speculative resource sharing is permitted, provided that all the data needed is available, and the shared operation is guaranteed not to stall.

Both of these conditions are necessary to ensure that deadlock does not occur. Deadlock may occur  
10 if a shared operation is locked into an instance that is waiting for a valid value for one of the variables, but the variable is dependent on the result of a different usage instance that is waiting (in vain) for the resource to become available.

15 FIG. 23 is a circuit that illustrates how an arbitration scheme may be implemented in accordance with the present invention. The arbitration scheme chooses left-most resource request 140 to be granted exclusive use of the shared resource. Every instance has an  
20 associated latch 142 that remembers whether that instance has ever been granted. Latch 142 is considered to be part of the usage instance's context, and hence the usage instance's reset signal 144 is able to reset it. A shared operation cannot be reset mid-execution, and  
25 therefore the usage instances' reset signals never need to propagate to the shared block.

FIG. 23 assumes that all usage instances have the potential to request the shared resource simultaneously. In practice, this will not be the case  
30 because data dependencies may result in two usage instances not being able to request the same resource simultaneously. In these cases, arbitration has the potential to be simplified. For example, usage instances that are guaranteed not to conflict may be ORed together.



The software-to-hardware compiler of the present invention may provide support for arrays. In one suitable approach, arrays may be implemented as a set of variables. Although this allows multiple read and write  
5 accesses per clock cycle, this may sometimes consume too many hardware resources. In this case, the software-to-hardware compiler may determine that it is more appropriate to implement an array using on-chip or off-chip RAM. RAM-based arrays are state-based, and can only  
10 be accessed in word-sized chunks. That is, extra circuitry may be needed to read and write to the arrays, and some mechanism is needed for ordering reads and writes.

All reads and writes to the same RAM must  
15 ultimately be arbitrated, treating the RAM as a shared resource.

The sequence of write operations affecting a RAM is important and must often be preserved. Between write operations, read operations may occur in any order.  
20 However, arrays that use different regions of a RAM do not need their reads and writes ordered with respect to each other.

This read and write operation ordering may be maintained by per-array token passing. One suitable  
25 embodiment of the present invention is to use the following additional wires for each array: Have\_done\_write, and Can\_do\_write. Have\_done\_write will indicate whether a write operation has to be performed (i.e., it is safe to do an intermediary read operation).  
30 The Can\_do\_write wire will indicate whether all intermediary read operations have completed, and that it is therefore safe to do a write operation. FIGS. 24 and 25 illustrate the use of these wires in arbitrating a read and write operation, respectively.

A sequence of reads and writes to a RAM-based resource are coupled such that the output wires of a read and write are coupled to the next read or write of the next operation. Before any reads or writes, Can\_do\_Write  
5 or Have\_Done\_Write are set to "1." As shown in FIG. 25, a write operation must not only wait for address and data values to be valid, and the control flow to be in the commit state, but also for the write enable wire to be "1." As can be seen in FIGS. 24 and 25 the write enable  
10 is only "1" when the previous write operation has completed, and all read operation since that previous write have completed. The Have\_Done\_Write signal is used to enable all reads between two writes.

The techniques described above are not limited  
15 to arrays and RAM accesses. It will be understood that any shared resource may be represented in this way. In general, a "write" operation may be defined as any operation that can affect the state of the resource (i.e., a mutable operation). A "read" operation may be  
20 defined as any operation that does not affect the state of the resource (i.e., a constant, or immutable operation).

In general, functions and operations may be executed in any order provided that the data needed by  
25 each function and operation is valid. However, there are instances when this will not result in the desired behavior. For example, file reads and writes need to preserve their ordering. In the case of files, the file is an example of a shared resource, and functions that  
30 operate on files must be annotated by the user to show that a shared resource is involved.

The software-to-hardware compiler of the present invention may support pointers. Pointers have the ability to read and write to one of a number of  
35 variables, arrays, or both.

For many software languages, it is possible to identify the subset of variables, arrays, or both to which a pointer can point to at any one time. All other variables or arrays are unaffected by writes through that  
5 pointer.

FIG. 26 shows how an indirect pointer read may be implemented in accordance with the present invention. In this example, the software-to-hardware compiler has determined that pointer variable (p) 164 can only point  
10 to one of three things: a variable (x) 161, a variable (y) 163, or an element in an array (a[]) 165. Demultiplexer 160 decodes which object the pointer is currently pointing to, such that value (\*p) 162 may be computed. Until pointer variable (p) 164 is computed,  
15 value (\*p) 162 remains uncomputed. Any array that may be referenced by variable (p) 164 must wait for variable (p) 164 to become computed, even if value (\*p) 162 does not end up reading from this array. This ensures that no array writes occur before the array is potentially read.  
20 Apart from arrays, all other variables are unaffected by indirect pointer reads.

FIG. 27 shows how an indirect pointer write may be implemented. In this example, the software-to-hardware compiler has determined that variable (p) 172  
25 can only point to one of three things: a variable (x) 171, a variable (y) 173, or an element in an array (a[]) 175. Variable (e) 174 is intended to be written to the variable or array pointed to by pointer variable (p) 172. Pointer variable (p) 172 pointer is demultiplexed by  
30 demultiplexer 170 so that the correct object can be updated. None of the candidate variables or arrays are deemed computed until the pointer value is itself computed, and this is ensured by array\_rdy block 176 and var\_rdy blocks 177.

The software-to-hardware compiler of the present invention may be used to implement conditional statements in hardware. For example, FIG. 21 shows how an IF statement may be synthesized in hardware in accordance with the present invention.

Conditional block 180 is used to evaluate a conditional expression to determine whether THEN block 184 or ELSE block 186 is executed. Conditional block 180 may also update variables -- a common feature in some programming languages (e.g. C, C++). "Decide" block 190 passes control flow to THEN block 184 or ELSE block 186. THEN and ELSE blocks 184 and 186 both have access to the value of all variables in the current environment, but may change these values in different ways (e.g. THEN block 184 may add one to a variable, whereas ELSE block may multiply another variable by five). The output of THEN and ELSE blocks 184 and 186 is the value of all the variables in the environments resulting from executing each. The output of IF-THEN-ELSE block 185 has a single value for each variable. Thus, "join" block 192 establishes which value is appropriate. This is done by determining which of blocks 184 and 186 was executed.

Speculation allows conditional block 180, THEN block 186, and ELSE block 184 to execute simultaneously, provided that the value of variables needed in each are computed. Flowjoin, joinctl, and join blocks 194, 196, and 192, respectively, determine the final values of variables 193 in the environment at the end of IF-THEN-ELSE block 185.

If a variable is unchanged by conditional block 180, THEN block 186, and ELSE block 184, then there will be a direct path from the value of the variable at the input and outputs of IF-THEN-ELSE block 185. If the value of such a variable is computed at the input of IF-

THEN-ELSE block 185, then it is computed at the output. This may allow the next block in the program to begin execution provided that the values of variables needed are computed. The specification for decide block 190 is outlined in the following table, and an illustrative implementation for decide block 190, in accordance with the specification, is shown in FIG. 29. It will be understood that the hardware shown in FIG. 29 is merely illustrative and that any other suitable implementation may be used. It will further be understood that the hardware shown in FIG. 29 relies on the "speculate," "commit," and "cancel" encoding previously defined.

DECIDE

| Control flow | Decision variable | Then-Control flow | Else-Control flow |
|--------------|-------------------|-------------------|-------------------|
| Commit       | Undefined         | Speculate         | Speculate         |
| Commit       | True              | Commit            | Cancel            |
| Commit       | False             | Cancel            | Commit            |
| Cancel       | Don't Care        | Cancel            | Cancel            |
| Speculate    | Undefined         | Speculate         | Speculate         |
| Speculate    | True              | Speculate         | Cancel            |
| Speculate    | False             | Cancel            | Speculate         |

The specification for join block 192 and flowjoin block 194 is outlined in the following table, and illustrative implementations for join block 192, flowjoin block 194, and joinctl block 196 are shown in FIG. 30. It will be understood that the hardware shown in FIG. 30 is merely illustrative and that any other suitable implementation may be used. It will further be understood that the hardware shown in FIG. 30 relies on the "speculate," "commit," and "cancel" encoding previously defined.

FLOWJOIN AND JOIN

| Then-<br>Control<br>flow | Then-Data  | Else-<br>Control<br>flow | Else-Data  | Joined-<br>Control<br>flow | Joined-<br>Data |
|--------------------------|------------|--------------------------|------------|----------------------------|-----------------|
| Commit                   | Undefined  | Don't Care               | Don't Care | Commit                     | Undefined       |
| Commit                   | Defined(X) | Don't Care               | Don't Care | Commit                     | Defined(X)      |
| Cancel                   | Don't Care | Cancel                   | Don't Care | Cancel                     | Don't Care      |
| Cancel                   | Don't Care | Speculate                | Undefined  | Speculate                  | Undefined       |
| Don't Care               | Don't Care | Commit                   | Defined(Y) | Commit                     | Defined(Y)      |
| Cancel                   | Don't Care | Speculate                | Defined(Y) | Speculate                  | Defined(Y)      |
| Don't Care               | Don't Care | Commit                   | Undefined  | Commit                     | Undefined       |
| Speculate                | Undefined  | Cancel                   | Don't Care | Speculate                  | Undefined       |
| Speculate                | Defined(X) | Cancel                   | Don't Care | Speculate                  | Defined(X)      |
| Speculate                | Don't Care | Speculate                | Don't Care | Speculate                  | Undefined       |

15           The software-to-hardware compiler of the  
present invention may be designed to cope with  
speculation that enables the execution of two or more  
loops in parallel under certain circumstances. Because  
speculation beyond a loop may be hampered by a context  
20 reset, a more speculation friendly approach may be  
adopted in which only the loop body is reset.

          When the loop exits, the resulting environment  
is used to initiate the next instruction. Any  
speculation that occurs before the loop exit must result  
25 in values that will exist in the final environment (i.e.,  
not the intermediary values from the loop).

          In one suitable approach, the software-to-  
hardware compiler recognizes what variables are loop-  
invariant, or value-predictable from the outset of the  
30 loop. Hardware is then generated based on these  
findings. For example, if an input of a block does not  
change value throughout the operations executed in that  
block, then the value of the output is computed at the  
input. In such a case, the software-to-hardware compiler

may optimize the hardware by allowing loops to execute in parallel assured that variables will not be erroneously overwritten. In the case of conditionals, a join may be avoided (i.e., after having speculatively executed an  
5 ELSE-part and a THEN-part) because the variable remained unchanged.

FIG. 31 shows how the software-to-hardware compiler may construct hardware that behaves as a WHILE loop. A WHILE loop that does not include jumps has only  
10 one entry and one exit point. WHILE loop 251 takes as input, the values of variables 253 from the current environment, and outputs the values of variables 257 in the new environment.

The body of WHILE loop 251 is a single  
15 block 258. The values of variables 253 at the input to block 251 are derived from one of two sources. If this is the first iteration of WHILE loop 251, then the values are derived from the values of variables 253 at the input to the WHILE loop. Otherwise, these values are derived  
20 from pertinent state 255. Override block 254 performs this selection. Pertinent state 255 holds the values of variables 259 from the previous iteration of the WHILE loop 251. Thus, the values of variables 259 at the output of loop body block 258 are used to update  
25 pertinent state 255. When pertinent state 255 is assigned, conditional block 261, and loop-body block 258 are reset such that the next iteration of values may be computed.

In one suitable arrangement, the hardware  
30 responsible for executing the body of a WHILE loop is shared between the iterations of the WHILE loop. Consequently, a fixed amount of hardware may be used to perform an unlimited number of iterations. The software-to-hardware compiler may also determine that loop-  
35 unrolling or similar static optimizations yield a better

performance in terms of hardware-area or speed. These optimizations may be implemented in the hardware generated by the compiler.

When the conditional expression in WHILE  
5 loop 251 evaluates to false, conditional block 261 causes  
decide block 263 to assign the values of variables to  
output 257 of WHILE loop block 251. Until this time,  
wires 265 containing the values of variables at the input  
to loop body 258 may not be the same as the final values  
10 intended for output 257 of WHILE loop block 251. Commit  
block 250 ensures that the values of the variables at  
output 257 of WHILE loop 251 are not computed until  
conditional block 261 determines that WHILE loop 251 has  
ended. The hardware-to-software compiler may determine  
15 that the value of some variables (i.e., values 265) are  
the same throughout WHILE loop 251. Variable values 265  
are deemed loop-invariant, and can be connected from  
input 253 of WHILE loop 251 directly to output 257.

Loop block 267, merge block 252, and commitctl  
20 block 256 are used to propagate the control flow through  
WHILE loop 251. As can be seen from the tables defining  
these blocks, WHILE loop 251 may itself be executed  
speculatively.

The specification for commit block 250 is  
25 outlined in the following table. Commit block 250 waits  
for the loop to terminate before committing to give  
values to loop variant variables. Illustrative  
implementations of commit block 250 and commitctl  
block 256 is shown in FIG. 32.



Commit

| Control flow<br>(Body) | Control flow<br>(Condition) | New Control<br>flow | New<br>Variables<br>enabled? |
|------------------------|-----------------------------|---------------------|------------------------------|
| Commit                 | Don't Care                  | Commit              | No                           |
| Cancel                 | Commit                      | Commit              | Yes                          |
| Cancel                 | Cancel                      | Cancel              | Don't Care                   |
| Cancel                 | Speculate                   | Speculate           | Yes                          |
| Speculate              | Commit                      | Commit              | No                           |
| Speculate              | Don't Care                  | Speculate           | No                           |

The specification for merge block 252 and override block 254 is outlined in the following table. Illustrative implementations of merge block 252 and override block 254 is shown in FIG. 33. Merge block 252 and override block 254 compute the flow-control and variables for the current loop iteration. Variables from the pertinent state override the input variables of loop block 258.

Merge and Override

| Pertinent<br>Control<br>flow | Pertinent-<br>Data | Original-<br>Control<br>flow | Original-<br>Data | Merge-<br>Control<br>flow | Override-<br>Data |
|------------------------------|--------------------|------------------------------|-------------------|---------------------------|-------------------|
| Commit                       | Defined(X)         | Don't Care                   | Don't Care        | Commit                    | Defined(X)        |
| Don't Care                   | Don't Care         | Cancel                       | Don't Care        | Cancel                    | Don't Care        |
| Speculate                    | Defined(X)         | Commit                       | Undefined         | Commit                    | Defined(X)        |
| Speculate                    | Defined(X)         | Commit                       | Defined(Y)        | Commit                    | Defined(X)        |
| Speculate                    | Defined(X)         | Speculate                    | Don't Care        | Speculate                 | Define(X)         |
| Don't Care                   | Undefined          | Commit                       | Undefined         | Commit                    | Undefined         |
| Don't Care                   | Undefined          | Commit                       | Defined(Y)        | Commit                    | Defined(Y)        |
| Don't Care                   | Undefined          | Speculate                    | Undefined         | Speculate                 | Undefined         |
| Don't Care                   | Undefined          | Speculate                    | Defined(Y)        | Speculate                 | Defined(Y)        |

The specification for loop body block 258 is outlined in the following table. Loop body block 258 decides when to store the pertinent state. When the loop condition fails, then loop body block 258 generates a cancelled control flow (i.e., the state from the body is invalid and must not be stored).

| Control flow<br>(Body) | Done  | Enable State save |
|------------------------|-------|-------------------|
| Don't Care             | False | No                |
| Commit                 | True  | Yes               |
| Cancel                 | True  | No                |
| Speculate              | True  | Yes               |

There are several simplifications that can be made to the circuit of FIG. 31. For example, override block 254 only chooses values from pertinent state 255 when the values held there are valid. The pertinent control flow signal is either "commit" or "speculate" and the input control flow may only override a speculative pertinent state. Thus, the pertinent control flow signal does not need to be stored; and values are only stored in pertinent state 255 when loop body block 258 asserts the done signal. Because of this latter simplification, the values in pertinent state 255 are either all computed or all uncomputed. Therefore, it is only necessary to store a single computed bit in pertinent state 255.

Taking this simplifications into account, the circuit of FIG. 31 may be redesigned to a circuit such as that shown in FIG. 34.

In general, a program will spend most of its time performing computations within loops. It is therefore important to make loops iterate as fast as possible. Two main factors that prevent loops from being executed faster are limited resources (i.e., parallelism)

and data dependencies. In hardware, parallelism is less of an issue than in software. Therefore, data dependencies are likely to be a larger source of performance inefficiency than limited resources.

5           As described thus far, loops iterate once the new values of all the loop variant variables are known. However, in some circumstances a performance gain may be achieved by partially starting the next iteration with a subset of the values currently available. This can be  
10 done through loop-pipelining.

FIG. 35 is a decision diagram that outlines the requirements for determining whether loop-pipelining is feasible for a particular loop. If it is determined at box 300 that conditional blocks and loop blocks can be  
15 split such that no splits occur across edges representing variables (i.e., only expressions), and data flow across the split occurs in one direction only, then a single loop can be decomposed into two parallel (i.e., pipelined) loops at box 302. If the requirements of  
20 box 300 are not met, then pipelining is not possible without possible detrimental consequences (box 304). FIG. 36 shows an illustrative simplified implementation of loop-pipelining.

FIG. 36 shows two loops 314 and 316, having  
25 loop body blocks, 310 and 312, respectively, that may have been decomposed from a single loop by loop-pipelining. One loop does not depend on values computed in the other. Thus, data flows in one direction. It may be possible to further decompose these loops by further  
30 loop-pipelining.

Data flow edges 318 and 317 may be implemented using FIFO's or any other suitable data structure, (i.e., one per edge). This offers the advantage that loops with variable loop iteration speed iterate at a speed closer

to the average iteration speed, rather than at the slowest speed.

In order to maintain the functionality of the original loop, each decomposed loop 314 and 316 sends and  
5 receives the correct amount of data flow information on every iteration. If inter-loop data flow is implemented using FIFO's, one particular embodiment of the software-to-hardware compiler may wait for all FIFO's to have pushed and popped in the loop, before commencing the next  
10 iteration.

Pipelines may be simultaneously reset by resetting all blocks, and all relevant FIFO's. Pipelines may be reset in this manner even after many decompositions.

15 With regard to nested loops, when a loop has been repeatedly decomposed, there will be many parallel sub-loops with dataflow between them. Each sub-loop may be treated as an individual block, and the dataflow dependencies may be ignored. This allows the next level  
20 sub-loop to be decomposed in a similar way, aided by the fact that fewer dependencies exist between blocks.

After decomposition, some loops may only depend on dataflow data (i.e., data-input independent loops). These loops can execute as soon as data becomes available  
25 on a dataflow edge. In one suitable approach, the entire program can be modeled as such a loop, in which external data arrives and leaves on dataflow edges.

Within a loop, some data paths will terminate at a dataflow edge (i.e., at a FIFO). A loop cannot  
30 iterate until all FIFO's have pushed/popped values, and sometimes a loop can be waiting for data to reach a dataflow edge. In these cases, the long paths that end at a dataflow edge can be decomposed (e.g., by adding intermediary dataflow edges). This results in groups of  
35 edges whose inputs are dataflow edges and whose outputs

are dataflow edges. These groups are loops. Hence, further loop-decomposition has occurred.

FIG. 37 shows an illustrative pop interface buffer 320 (to be described later) that may be generated in hardware by the software-to-hardware compiler. A pop interface may be used to read data from a dataflow edge in a loop-pipeline. FIFO 320 may store the value of a given variable (or a set of variables). FIFO 320 may also store whether the value for the variable had been computed. For some instances, a variable is always computed before its value is stored on a FIFO, and so this extra bit may not always be required.

FIFO 320 may sometimes be empty. In these cases, the value of the variable that is supposed to be stored in FIFO 320 is deemed not to be computed, and the block in which FIFO 320 resides does not indicate that it has completed (and hence cannot be reset). FIG. 37 shows illustrative hardware that may be generated by the software-to-hardware compiler to achieve this.

If a FIFO is contained in a loop-body block, then the loop requires that block to be reset upon every iteration. Upon loop-reset, the FIFO is not reset, but simply "popped." That is, the current value on the FIFO is discarded, and the next value on the FIFO takes its place. This effect could be achieved by incrementing read-counter 325.

FIG. 37 shows FIFO 320 acting as an input. FIFO 320 supplies the current context with variable 322 taken from the head of FIFO 320. When all FIFO's have supplied a variable into the current context, and the execution frontier has reached all the context outputs, then the context can be reset. A FIFO does not reside in the context of the loop body but in the context of the loop itself, hence a FIFO is reset when the entire loop

is reset. However, a loop body reset (e.g., loop body reset 324) pops a value from all the FIFO's used by it.

FIG. 38 shows how a push interface may be implemented. The context is not completed until the push  
5 has been performed (i.e., assuming the buffer is not full). Loop body reset 332 completes all pushes. FIFO 330 is reset by the reset for the entire loop.

A reset signal for a block may be asserted only when execution of that block is complete. That is, a  
10 reset signal is asserted when all FIFO's have either supplied (i.e., popped) valid data or are ready to absorb (i.e., push) valid data.

Once a block in a loop-body has completed execution, the outputs of the block are latched. The  
15 block is then reset and restarted. For a decomposed loop, only a subset of this latched data is required to be looped-back. The remainder is sent to the next loop in the decomposed sequence. The loop block cannot restart until all required data from previous loops in  
20 the decomposed sequence is available, and all subsequent loops have read the previous data.

If a single register is used to buffer between loops in a decomposed sequence, then there may be many instances when a loop will stall, waiting for data. The  
25 software-to-hardware compiler may determine that it is more appropriate to insert FIFO buffers instead of single registers, in order to minimize stalling. This is especially beneficial if loops take an unpredictable amount of time in each iteration.

30 In one embodiment of the software-to-hardware compiler of the present invention, FIFO buffers may be inserted throughout a hardware design -- not just in loops.

Most processor or system buses can be made to  
35 behave as FIFO buffers. Thus, the software-to-hardware

compiler may decide at a late stage to split the design between two or more hardware units in a system. An example of a hardware unit may be other programmable logic devices, or even other software processors. In this way, the software-to-hardware compiler may split a design between software and hardware, guaranteeing the functionality remains the same, but potentially improving performance, cost or size. This split may be done either automatically or with some user input.

FIG. 12 introduced a simple version of a block. Taking into account the additional control signals that may be used to implement features such as loops, sharing, pipelining, etc., FIG. 39 shows a more robust block 340. Reset signal 342 may be needed for loops and sharing. Reset signal 342 resets the context of block 340 on the next clock cycle. Only state-holding elements need to be reset.

Done signal 344 is asserted when no more variables in the output environment of a block will be computed. A reset cannot be applied until a block's done signal has been asserted.

Reset circuitry may incur large propagation delays for large designs. To reduce these delays, the compiler may insert extra latches in the reset paths provided that it can be guaranteed that the reset signal will reach all state elements in a block before the next valid data does.

The software-to-hardware compiler may ensure that, prior to optimization, every set of data wires has an associated control wire. The hardware will operate correctly, provided that the control wire goes high after the correct data values have been asserted. This allows registers to be added to data wires and control wires, provided that the number of registers added to a control wire is at least as many as the highest number of

registers added to any data wire.

The approximate clock rate at which the hardware circuit will operate is determined by calculating the longest asynchronous path between any two registers. In order to maximize the clock rate of the hardware, the software-to-hardware compiler may ensure that all paths between registers do not exceed a particular length. The software-to-hardware compiler may determine whether to insert extra register stages to improve the clock rate. The number of register stages together with the clock rate determine the overall system performance.

It will be understood that the features and details of the software-to-hardware compiler described herein are merely illustrative and, to some extent, simplified. It will be understood that these, and other suitable features and details may be scaled to accommodate large programs.

To illustrate how the software-to-hardware compiler of the present invention converts a sequential program into hardware, the following C-code for determining the greatest common divisor (GCD) of two numbers will be compiled into hardware:

```
While (x != y) {  
    If (x > y)  
        x -= y;  
    Else  
        y -= x;  
}
```

Using the software-to-hardware compiler, this program may be translated into the block shown in FIG. 40. Block 350 contains the while loop-body shown above. Inputs 352 and 354 translate to "x" and "y," respectively. Outputs 356 and 358 translate to the GCD of "x" and "y,"



which are provided when "x" equals "y" according the end-condition of the while loop.

Using the transformations for WHILE loops and IF blocks, the resulting hardware is shown in FIG. 41.

5 The GCD program is composed of several nested blocks, which in this case are contained within a single outer block -- WHILE loop 360. WHILE loop 360 is based on the implementation of a WHILE loop as was shown in FIG. 37. In the case of the GCD program, there are only two  
10 variables the values of which matter -- "x" and "y." Thus, input environment 362 and output environment 364 of WHILE loop 360 contain only these values.

A WHILE loop contains two sub-blocks, one for the conditional expression, (i.e., conditional block 374,  
15 which in this case computes "x != y"), and one for the body of the loop. In this case, the loop body is itself composed of several sub-blocks, the outer-most of which is a single IF block. Using the implementation for an IF block as shown in FIG. 21, loop body 365 may be  
20 decomposed into several sub-blocks. That is, Loop body 365 may be decomposed into blocks that include one for the conditional expression of the IF (i.e., conditional block 366), one for the THEN expression (i.e., THEN block 368), and one for the ELSE expression  
25 (i.e., ELSE block 370). IF block 365 only requires the values of two variables in its input and output environments -- "x" and "y". The values at the output of IF block 365 represent the values of "x" and "y" at the end of a WHILE loop iteration and are passed into  
30 pertinent state register 372 so that they may be used on the next iteration of the WHILE loop. The software-to-hardware compiler recursively transforms blocks until the entire program can be expressed in terms of interconnected operations and primitive hardware gates.

In the case of the GCD example, FIG. 42 shows the resulting hardware.

Some control blocks have not been reduced to gates for clarity. As can be seen by in FIG. 42, the values for the variables "x" and "y" are given to GCD block 378. Because the pertinent state is initially unassigned, override blocks 380 and 382 select the original input values of the variables "x" and "y" to be used in the first iteration of WHILE loop 378. These values are compared using the "not-equals-to" operator 384, which feeds decide block 386. The decide block 386 determines whether the current values of "x" and "y" are the final values of "x" and "y" at the output of WHILE loop block 378. If not, then the control flow signals commit to the loop body block 388. By coupling the wires holding the values of "x" and "y" to subtractors 390 and 392, join blocks 394 and 396 may select whether ("x-y" and "y") or ("x" and "y-x") should be used as the updated values of "x" and "y," respectively. These values are assigned to pertinent state register 398 for the next iteration of the WHILE loop.

A few extra control wires are needed to complete the circuit in FIG. 42, such "loop-block-done" and "loop-block-reset" signals. The exact hardware needed for these signals depend on how many registers or FIFO's have been inserted on any of the bold black lines. The software-to-hardware compiler may decide to implement any number of registers or FIFO's throughout the design, without affecting the entire circuit operating as a GCD. The software-to-hardware compiler may determine the number of register delays in computing the loop body, and assert the "loop-block-done" signal after this time. If no registers are inserted, the "loop-block-done" signal may be asserted immediately. When the "loop-block-done" signal is asserted, the pertinent state register 398 may

latch the values of "x" and "y" on the next clock cycle,  
as well as reset the loop-body. The "loop-body-reset"  
resets all registers within the loop-body. The software-  
to-hardware compiler may determine that sometimes only a  
5 subset of registers need to be reset.

Another aspect of the present invention is  
directed toward communication between hardware devices,  
between software devices, and between software and  
hardware devices. Hardware devices include any suitable  
10 device implemented in hardware, such as programmable  
logic resources. Software devices include any suitable  
device that executes software constructs, such as  
microprocessors. It will be understood that a device may  
include multiple devices coupled to one another.

15 According to the present invention,  
communication between devices can be implemented using a  
FIFO buffer. For clarity, a FIFO will be regarded as  
having two separate interfaces: one which is responsible  
for adding data elements to the FIFO (i.e., a "push"  
20 interface), and one which is responsible for reading and  
removing data elements from the FIFO (i.e., a "pop"  
interface). A push interface also indicates when it is  
able to accept a new data element, and a pop interface  
indicates when it is possible to read and remove a data  
25 element. It will be appreciated that the present  
invention may be implemented using data structures other  
than FIFO's. For example, stacks, priority queues, or  
any other suitable data structure or selection algorithm  
may be used in accordance with the present invention.

30 In hardware, one suitable embodiment of a FIFO  
interface is a set of data wires and control wires. Data  
wires may be used to carry a binary representation of the  
data element to be pushed or popped. One control wire  
indicates whether a push or pop can be completed, and  
35 another control wire indicates whether a push or pop

should be attempted. The behaviour of a FIFO when a push or pop is attempted but cannot be completed is undefined, and is regarded as an illegal use of the interface.

In software, a FIFO interface may be  
5 implemented as a single function call or set of function calls (e.g., a set of C-functions, C++-methods, or any other suitable construct in C, C++, Modula, JAVA, LISP, etc.). At minimum, a push interface may be a single function with a single argument and no return value. The  
10 argument of such a function is the data element or a pointer to the data element required to be added to the FIFO. Such a function would only complete when the FIFO has successfully accepted the data element. At minimum, a pop interface in software may be a single function with  
15 no arguments and a single return value. Such a function would only complete when the FIFO has successfully removed a data element, and the return value would be a copy of the data element removed or a pointer to a copy of the data element removed.

20 In one suitable arrangement, one FIFO interface may be instantiated in a hardware device and the corresponding FIFO interface may be instantiated in a software device. According to the present invention, these interfaces are used as though there was a single  
25 FIFO buffer between the devices.

In another suitable arrangement, both FIFO interfaces may be instantiated in hardware. That is each interface resides in a different physical hardware device.

30 In general, multiple independent FIFO buffers may be needed between devices (i.e., software devices, hardware device, or both). Although each FIFO buffer has only one push interface and one pop interface, the size of data elements in each FIFO may be different.

FIG. 43 shows how several FIFO's buffers communicate between two separate devices. The entities in FIG. 43 are intended to replicate the functionality of five separate FIFO buffers. It will be understood that any suitable number of FIFO buffers may be used and that FIG. 43 is merely illustrative. Each FIFO buffer has two interfaces available to the user who will ultimately be unaware of the channel multiplexer and channel demultiplexer. For example, a push interface for buffer 400 and a pop interface for buffer 410 make up the interfaces needed for a single FIFO buffer. FIFO push interfaces for buffers 400, 402, and 404, and FIFO pop interfaces for buffers 406, and 408 reside on one device, and FIFO pop interfaces for buffers 410, 412, and 414, and FIFO push interfaces for buffers 416 and 418 reside on the other device.

These buffers will be referred to herein as "push interface buffers" and "pop interface buffers." Interface buffers reside on the same device as their respect interfaces. The number of data elements that each of these buffers can hold is not important, except that a push interface buffer needs to know the maximum number of data elements that its corresponding pop interface buffer can hold. For purposes of illustration and not by way of limitation, it will be assumed that all buffers hold the same number of data elements.

Hardware resources, software resources, or both may be added for transporting data elements from push interface buffers to pop interface buffers. In one suitable embodiment, two separate channels are used to transport data elements, one in each direction. This is illustrated in FIG. 43 in which channel 433 is a unidirectional channel that carries data elements from push interface buffers 400, 402, and 404 to pop interface buffers 410, 412, and 414, respectively. Channel 432 is

also a unidirectional channel that carries data elements in the opposite direction -- from push interface buffers 416 and 418 to pop interface buffers 406 and 408.

In another suitable embodiment, a single  
5 bidirectional channel may be used by time-multiplexing data being sent in either direction.

Each push interface buffer attempts to send a number of data elements to the corresponding pop interface buffer. When the pop interface buffer pops a  
10 number of data elements, an acknowledgment is sent to the push interface buffer (e.g., in acknowledgments 434 or 436). Until an acknowledgment is received, the push interface buffer assumes that all unacknowledged data elements still reside in the pop interface buffer. Thus,  
15 the push interface buffer will not send a data element that might overfill the pop interface's buffer.

As shown in FIG. 43, acknowledgments 436 are sent to push interface buffers 416 and 418 by pop interface buffers 406 and 408 via channel multiplexer 420  
20 to channel multiplexer 426. Likewise, acknowledgments 434 are sent to push interface buffers 400, 402, and 404 by pop interface buffers 410, 412, and 414 via channel multiplexer 426 to channel multiplexer 420.

In one suitable approach, each acknowledgment  
25 may be used to acknowledge that a fixed number of data elements have been popped. This number may vary from FIFO buffer to FIFO buffer but should be constant during channel operation.

In one embodiment, a cluster of data elements  
30 may be sent across the channel of the present invention, where the size of clusters may vary during channel operation. The push interface buffer keeps track of the sizes of each cluster sent, and the pop interface keeps track of each cluster size received. Each acknowledgment  
35 may be used to acknowledge when a cluster of data

elements have been popped.

The present invention requires that a channel multiplexer (e.g., multiplexers 420 and 426 of FIG. 43) combine packets from the pop interface buffers onto a single channel buffer. Extra hardware resources, software resources, or both may be used to transport packets from this single channel buffer to a channel demultiplexer.

Channel demultiplexers 422 and 424 determine to which pop interface buffer an incoming data element is destined. For every FIFO buffer whose push interface buffer resides on the same device and whose pop interface buffer resides on the same device, a unique identifier, referred to herein as a "FIFO ID," is assigned. When a push interface buffer sends a number of data elements, they are preceded with a header that indicates the FIFO ID as well as the number of data elements to follow. The sequence of a header followed by a number of data elements is referred to herein as a "frame." Channel multiplexers 420 and 426 are responsible for storing sequences of frames in their respective channel buffers, and channel demultiplexers 422 and 424 are responsible for decoding these frames.

FIG. 44 shows a typical sequence of packets, each packet 452 having a header 454 and data items 456. Header 454 may include bit 466 indicating whether header 454 is valid. Bits 457 indicate the FIFO ID associated with packet 452. Bits 458 indicate the number of data items in packet 452. Acknowledgments 460 and 462 may be sent as part of frame header 454. As shown in FIG. 44, the number of FIFO ID's is limited to 256, and frame header 454 allows up to two acknowledgments 460 and 462 to be sent. A FIFO ID of "0" may be used to indicate when no acknowledgment needs to be sent. When no data needs to be sent over a channel, a special dummy

frame header may be used if the channel requires that data be sent every clock cycle. Bit 464 in frame header 454 may be used for this purpose.

Channel 450 is shown as being 32-bits wide.

5 However, it will be appreciated that any suitable width channel may be used. Protocols may be devised that may accommodate smaller or larger data widths. It will be further appreciated that a fixed channel size does not necessarily imply that the interface buffer must be  
10 padded (or packed) into that fixed size.

The communication interface of the present invention may be implemented using any suitable data transport medium. In general, these transport media are in the form of a direct interconnect, shared memory, or a  
15 bus.

A direct interconnect transport medium couples two devices with point to point wires. For example, frames may be sent directly to the channel demultiplexer across the wires. If the width of the FIFO does not  
20 match the number of wires in the transport medium, then data serialization may be used.

A shared memory transport medium couples two devices with a dual-port memory (or a time-multiplexed single port memory). For example, the channel buffer in  
25 the channel multiplexer may be implemented using the shared memory to store frames as shown in FIG. 45. The channel multiplexer would be responsible for pushing frame data onto shared memory, and the channel demultiplexer could read directly from the same memory.

30 FIG. 45 shows buffers 730 and 732 in shared memory partitioned between software devices 742 and hardware devices 744. Each of devices 742 and 744 is responsible for a "head" and "tail" pointer used to push and pop data elements onto buffers 730 and 732. That is,  
35 software devices 742 use head and tail pointers 738



and 734 and hardware devices 744 use head and tail pointers 736 and 740. To avoid having the buffers popped when empty, or pushed when full, special acknowledgments are inserted in buffers 730 and 732 by one device to  
5 update the other device of the current value of the head pointer. These updates are stored using assumed head pointers 746 and 748. The values are conservative assumptions because by the time they are read, the actual head pointer may have changed.

10           FIG. 46 illustrates a bus-based channel implementation. In particular, FIG. 46 shows how a channel multiplexer and channel demultiplexer may be implemented using a PCI bus transport medium between a hardware device and a software device. It will be  
15 understood that any other suitable bus may be used. Software input buffer 714 and software output buffer 718 represent buffers contained in memory space that is accessible to both devices 702 and 704. Often, for speed of access, buffers 714 and 718 are located in the  
20 software device physical memory. Likewise, hardware input buffer 712 and hardware output buffer 708 represent buffers stored on hardware device 704.

          In one suitable approach, frames may be communicated from hardware device 704 to software  
25 device 702 in super-packets. As shown in FIG. 47, super-packet 670 includes a super-packet header 672, packets 676, markers 684, and an end\_marker 680. Markers 684 include indices to entries 676 in super-packet 670 that could be mistaken for an end\_marker.  
30 Markers 684 will depend on data 676 within packets, and thus need to be determined for every super-packet 670. When super-packet 670 has been sent, no further super-packets can be sent until super-packet acknowledgment 716 is received back from software devices 702.

Acknowledgment 716 is sent once the entire super-packet has been processed by software devices 702.

Super-packets are beneficial because they allow for the use of polling as opposed to interrupts and interrupt handlers. Provided that data is communicated on a regular basis between hardware device 704 and software device 702, this leads to a dramatic improvement in performance on systems where the software device can access software input buffer 714 at speeds much faster than the bus transfer speeds, or when a large number of relatively small super-packets need to be sent in succession.

Super-packet header 672 is used to detect the presence of end\_marker 680. If the location after the location indicated in size bits 674 in super-packet 670 is not an end\_marker, then the system is able to detect that the transfer of super-packet 670 to software input buffer 714 has not yet completed. When a super-packet transfer has completed, and the packets within have been transferred to their respective buffers, software devices 702 are able to use markers 684 within the super-packet to set all locations that could be mistaken for an end\_marker to a value that is not an end\_marker. Thus, the software input buffer is left in a state where a new super-packet is ready to be received. Initially, however, the software input buffer must contain values that are not end\_markers.

Super-packets may always be written from the start of the software input buffer. Thus, the super-packet header is always at the same location in memory.

Super-packets are not needed for data transfers from software to hardware because the hardware is usually informed directly when a bus transfer is initiated and completed. However, software device 702 ensures that it does not overfill hardware input buffer 712, and so must

sometimes wait for special acknowledgments from hardware device 704 indicating that packets have been popped from the hardware input buffer. These acknowledgments may be carried in super-packet header 672, using bits 686.

5 Transfers of packets from software to hardware may be initiated from software. In one embodiment of the present invention, this can be achieved by writing to a special memory location reserved by the bus for the hardware device.

10 The present invention ensures that deadlock cannot occur by always allowing software output buffer 718 to send an acknowledgment of a super-packet in software input buffer 714, even though hardware input buffer 712 may be full. So as not to overfill hardware  
15 input buffer 712, hardware device 704 intercepts acknowledgments before they are added to hardware input buffer 712, and does not add them to the header input FIFO.

FIG. 48 shows how multiple FIFO's 646 may  
20 reside in a software device's memory space. On a multi-tasking operating system, multiple processes 654, 656 and 648 may be pushing or popping data from FIFO's 646. An arbitrator 644 is an embodiment of a software channel multiplexer and demultiplexer, and as such, keeps track  
25 of FIFO acknowledgments (as shown by boxes 650 and 653, which reside in local memory), and may be responsible for sending data elements to and from channel buffers 640 and 642 which correspond to buffers 714 and 718 shown in FIG. 46. In one suitable embodiment of the present  
30 invention, arbitrator 644 may be implemented as a device-driver.

Channel multiplexers and channel demultiplexers for FIFO interface buffers residing on a hardware device may be implemented based on what is shown in FIGS. 49-55.

FIG. 49 defines a "stream" 470 using a set of wires. Stream 470 is a set of wires 474 on which is sent serialized frame data. An additional type wire 472 is used to indicate when data wires 474 hold a frame header.

5 Stall wire 476 is used to indicate that the channel multiplexer buffer is full, and hence all transmission of frames must stall. Frame data sent to a channel demultiplexer can never stall because the communication system of the present invention ensures that the

10 destination pop interface buffer can never be full.

FIG. 50 shows how data elements from push interface buffer 522 may be constructed into a frame (complete with frame header). Finite state machine (FSM) 524 may create a frame of data elements from push

15 interface buffer 522. FSM 524 may require a number of signals shown in FIG. 50 to determine whether any data elements exist on push interface buffer 522, and if so, sends a number of data elements as a single frame. As illustrated in FIG. 50, it is assumed that the remote pop

20 interface buffer can hold at least as many data elements as buffer 522, and that no more frames can be sent until an acknowledgment has been received. The arrangement of FIG. 50 results in a single stream. If no packet data can be sent, then the stream data sent is a packet

25 composed of only an invalid packet header.

FIG. 51 shows how two streams may be merged. Arbitrator 506 may implement a fair algorithm that switches between packets on stream 500 and packets on stream 502, depending on which has a valid frame header

30 and/or which stream was chosen last time. Arbitrator 506 uses the AND and OR gates as a multiplexer to choose which stream is sent, and which is stalled.

Cascading the stream-merging circuits allows multiple FIFO streams to eventually merge into a single

35 stream 504.

If a large number of streams need to be merged, then it may be desirable to insert pipelining into the streams because this reduces the propagation delay between registers, thus allowing the stream-merging to run at higher clock frequencies. FIG. 52 shows how a register 478 may be inserted into a stream. FIG. 53 shows how, with additional hardware, a register 480 may be inserted into a stream while ensuring that the stream does not stall when register 480 holds an invalid frame-  
10 header.

FIG. 54 shows how a stream may be modified to include pending acknowledgments before being sent to the channel buffer (or I/O pins in the particular arrangement shown in FIG. 54). Acknowledgments from pop interface  
15 buffers may be queued on a separate buffer 574. In this embodiment of the present invention, two acknowledgments may be sent with every frame, so extra hardware 570 and 572 is used to implement a deserializer. If no acknowledgments are pending, the FSM is responsible for  
20 resetting the acknowledgments to a special value. Because only one pop interface buffer can be receiving a frame at any one time, only one acknowledgment can be added to acknowledgment FIFO 574 at any one time.

FIG. 55 shows how a channel demultiplexer may  
25 be implemented in hardware. Frames are received on I/O pins 600 or from a channel demultiplexer buffer, and FSM 608 is used to read each frame header and pass on its contents to the relevant pop interface buffer. Acknowledgments 606 are also extracted from the frame  
30 headers so that the pop interface buffers on this device can receive them.

Thus, a software-to-hardware compiler is provided. One skilled in the art will realize that the present invention can be practiced by other than the  
35 described embodiments, which are presented for purposes

of illustration and not of limitation, and that the present invention is limited only by the claims which follow.